Audio Programming Environment - Manual

Introduction

Welcome to the manual for the APE program, Audio Programming Environment. APE is the brainchild of me, having struggled to learn to write DSP code in a practical manner. I grew tired of spending most of my time writing interfaces, frameworks, GUI's and whatnot, managing different libraries, dependencies etc. for very small plugins until I ultimately decided that it would be better to abstract the whole package away - enter APE.

APE is a comparatively small program that hosts user-written code and integrates it into the signal path of your host program. It provides a small API for making simple controls, like knobs and buttons to adjust variables inside your code, and to provide certain system information like tempo, channels etc. together with a console and an editor, that allows easy development and testing.

APE's intended use is the development process - testing out and fine-tuning algorithms in an efficient manner before integrating it into your primary project. It allows for on-the-fly compilation and testing. Also, it's intended to provide a simple introduction to writing DSP code in a manner where you don't have to worry about anything but the actual relevant code. It is therefore an ideal tool for teaching/educational projects and demonstrating small algorithms and / or effects.

I hope that someday it will evolve into a userbase-provided library of effects with a working implementation, with a design that allows to easily use the code in your own projects. Many great sites on the internet provide a lot of great examples, though they are mostly scattered and / or incomplete.

Features

- System-level exception handling to avoid crashing your host on small errors like integral division
- Optional floating-point unit exceptions for easier `NaN`/`Inf` tracing
- Console with logging
- Shipped C compiler (Tcc4Ape)
    - Over 20 working examples
- Ability to interface to any compiler installed on the computer, write in any language you want!
- Scintilla-based editor with syntax highlighting
- Small GUI library for creating controls
- Control and program events
- Extensive configuration file
- Flexible - supports any language that can provide C-bindings through a small compiler API and some settings in the configuration file

WIP features

- Adding support for compilation of programs to self-contained plugins
- Adding support for presets
- Integrated projects and tabbed editor
- Better GUI and better graphics (for controls, as well)
- ~~Cross platform support~~ nearly there!
- More languages supported, specifically Python

User interface

I would say it's pretty self-explanatory, but for good sake lets include a description:

- Console - Opens/closes the console window
- Compile - Tells the compiler to prepare the code it receives for execution
- Activate/disable - Activates the plugin and, if no error, uses it in the processing chain
- Editor - Spawns the editor window
- About - tells a nice story
- Use protected buffers - passes protected memory that identifies buffer over-/under-run problems
- Enable FPU exceptions - this toggle sets the FPU to throw exceptions on following events:
    - `_EM_INVALID | _EM_DENORMAL | _EM_ZERODIVIDE | _EM_OVERFLOW | _EM_UNDERFLOW`
    - These exceptions are caught by APE, however since audio processing usually is threaded, this might cause problems in either your host and/or other plugins. APE cannot prevent this, so you should not use this by default.
    - This feature is still pretty buggy.
- The text at the bottom right is the status text - will let you know what is happening
- `Accps` is the amount CPU cycles it took for the plugin to process a single sampleframe - that is, one sample for each channel.
- The rest of the unused space is used for controls/knobs

Editor

The editor is a simple text editor based on the open source *Scintilla Scilexer*[i] control.

It contains the usual stuff, including file open/save functionality.

Configuration

The configuration file resides in the root of the APE directory and is called *config.cfg*. APE uses the open source library *libconfig*[ii] to read and parse the configuration file. The syntax is easy and structured, and can be learned from the linked homepage. Currently, the file consists of two segments, the application- and language settings.

Application settings

This section contains the global application settings.

```
application:
{
        log_console = false;
        use_buffers = true;
        use_fpe = false;
        num_channels = 2;
        force_single_precision = true;
        greeting_shown = true;
        unique_id = 4280389;
        ui_refresh_interval = 80;
        console_std_writing = false;
        autosave_interval = 60;
        render_opengl = false;
        use_tcc_convention_hack = false;
}
```

- `log_console` - boolean - logs the output of the console into APE/logs/
- `use_buffers` - boolean - toggles the *Use Protected Buffers* switch in the UI
- `use_fpe` - boolean - toggles the *Use FPU exceptions* switch in the UI
- `num_channels` - integer - sets APE to use the amount of channels (ignored)
- `force_single_precision` - boolean - if set to false, APE will default to the highest possible bitdepth in the audio streams (ignored)
- `greeting_shown` - boolean - if set to false, APE will show a welcoming message on startup
- `unique_id` - integer - the id APE will use to identify itself (4280389 = ascii constant 'APE')
- `ui_refresh_interval` - integer - miliseconds, time between each redraw. This can be set very high if you dont care for quickly updating plots or displays (note, this will not affects normal redraw of the gui on events)
- `console_std_writing` - boolean - logs the output of the console to stdout
- `autosave_interval` - integer - amount in seconds between each autosave. Note that autosave only occurs if the code document has changed.
- `render_opengl` - boolean - renders the gui using opengl. Can give performance increase on OSX
- `use_tcc_convention_hack` - boolean - toggle this if you're having problems with knobs on x64

Editor settings

This section controls the editor.

```
editor:
{
        hkey_save = "cmd+s";
        hkey_open = "cmd+o";
        hkey_new = "cmd+n";
}
```

This will allow you to change the hotkeys of the editor. `cmd` is `command` on OSX and `control` on Windows.

Language settings

This section contains the language settings. It has the following structure:

```
languages:
{
        default = "c";
        default_file = "examples/template.c";

        <language_n-specification>
}
```

The `default` setting is the default language that the editor will select. This is mostly deduced from the `default_file` setting, which is what the editor will open on a fresh load.

The language specification instructs the program on what to with a specific language. APE supports any number of language specifications, as languages and compilers are selected and identified from the file extensions. They have the following structure:

```
<language-name>
{
	extensions = ("");
	compiler:
	{
		name = "";
		path = "";
		arguments = "";
		exports:
		{
			<export_n> = <export_n-name>;
		}
	}
	lexer:
	{
		scilex_number = -1;
		keywords:
		{
			color = "";
			words = "";
		}
		types:
		{
			color = "";
			words = "";
		}
		userdefined:
		{
			color = "";
			words = "";
		}
	}
}
```

`language-name` is the name of the language. `extensions` is a list of strings, comma-separated, that denotes what file extensions the editor can open and what language they are to be associated with. `name` is the name of the compiler (irrelevant, but for debugging), `path` is the relative path to the module that is the compiler, and `arguments` is a classic commandline that is passed to the compiler.

`exports` is a group of aliases for the names APE look up. See the section *Compiler API* for more info. You can change the name APE looks up by using this pattern:

```
GetSymbol = "x";
```

Where "x" is a string representing the usually decorated name.

The `lexer` part is mostly syntax-highlighting. `scilex_number` tells the Scilexer control to set a certain lexer. 3 is for instance *C++*. A list can be found in the file *scilexer.h* (in the APE src directory). *keywords/types/userdefined* allows to group and add keywords to the syntax highlighting.

Source code

The source code for both the APE plugin and the compiler Tcc4Ape and syswrap can be found in the */src/* directory. See */licenses/* for details on using it, and *readme* for compiling instructions.

Compiler API

APE is designed to be extensible and open. It supports any language so long as it has a compatible compiler. This way, you can write in your preferred language.

When APE is requested to compile and load the current script in the editor, it requests the editor for a project info struct. The editor composes this struct with information about the current relevant files and identifies the language based on the file extension of the main file.

APE passes this information on to the code generator inside APE. The code generator is responsible for communication between APE and the script. The code generator looks up the desired language in the configuration file, based upon the language ID (derived from the file extension). Here it collects information about the compiler settings to be used.

Using the supplied path, it loads a module (DLL on windows, Bundle on Mac, SO on *nix). APE will then try to retrieve pointers using the exported names in the configuration file for all the required functions.

Here's a list of what APE expects to be exported (if nothing is otherwise defined):

```
GetSymbol
CompileProject
SetErrorFunc
ReleaseProject
InitProject
ActivateProject
DisableProject
GetState
AddSymbol
ProcessReplacing
OnEvent
```

As you probably can see, all communication between the script and APE is dynamically loaded and done with several layers of indirection. This means that as long as the above functions returns correctly, APE doesn't have to know anything about any specific language or whatever. Indeed, the compiler may host a client process to do the actual processing, and pass buffers using shared memory.

Compilation, activation, idle and run process - function reference

To give an complete idea of what happens while creating a function reference, let's do it chronologically. All of the mentioned function takes a pointer to a `CProject` struct. All of the functions must return a `Status`, identifying success. This struct contains information about all of the relevant files needed to compile the project, settings for the compiler, state of the code and lastly, space for the compiler to keep its relevant instance data. Specific information about all of this is found in the file *APESupport.h*

Once the codegenerator has loaded the compiler module and ensured all relevant functions exists, it calls the following function:

```
Status _cdecl CompileProject(CProject * p, void * op, errorFunc_t e);
```

The compiler must store the `op` parameter when it uses the callback error function `e`:

```
typedef void (_cdecl * errorFunc_t)(void*, const char *);
```

`op` is to be used as the first parameter, the second being an error message back to APE. This callback function prints text to the associated console (identified through the `op` parameter). Example:

```
typedef void (_cdecl * errorFunc_t)(void*, const char *);

struct my_data
{
	...
	errorFunc_t errorBack;
	...
	void * op;
	...
	void printError(const char * msg) {
		if(errorBack && op)
			errorBack(op, msg);
	}
};

Status _cdecl CompileProject(CProject * p, void * op, errorFunc_t * e) {

	my_data * data = new my_data;
	data->op = op;
	data->errorBack = e;
	p->userData = data;
	...

	return Status::STATUS_OK;
}

Status _cdecl ActivateProject(CProject * p) {
	my_data * data = reinterpret_cast<my_data>(p->userData);

	...

	data->printError("Error activating project!");
	return Status::STATUS_ERROR;
}
```

The `CompileProject()` function is a request from APE to compile the current files in the `CProject` struct. The compiler is not supposed to link the code in this step.

If the return value of `CompileProject()` is `Status::STATUS_OK`, APE later issues a call to `InitProject()`.

```
Status _cdecl InitProject(CProject * p);
```

`InitProject()` finalizes the link time code generation and ensures the compiled code is executable. InitProject is assumed to return `Status::STATUS_OK`. This marks the end of the compilation process.

After this, the compiled code is assumed to be in either an active or deactivated state, starting with deactivated. When the user tries to activate the project, the following function is called:

```
Status _cdecl ActivateProject(CProject * p);
```

This function is supposed to initiate any runtime libraries, constructors and whatnot the compiled code requires and ensure that the compiled code is (in)directly executable - this is what happens next. The compiler is allowed to run user-written code in this step. If this step is successful (the return shall be `Status::STATUS_READY`), APE assumes that the script is activated and ready to process audio through the following function:

```
Status _cdecl ProcessReplacing(CProject * p, float ** in, float ** out, int
sampleFrames);
```

The return of the function should be `Status::STATUS_OK`. Any other return directly changes APE's internal status. For example, returning `Status::STATUS_DISABLED` will initiate disabling process. Returning `Status::STATUS_ERROR` will discard everything in the project to avoid compromising host stability. This is true for all of the mentioned functions here.

The purpose of this function is to give the script the ability to modify the audio stream how it seems fit, *however* it must respect that the output should be filled with values, modified or not. The size of the buffer is specified in `sampleFrames`. The amount of buffers is usually two (stereo), however changing settings in the configuration file can alter this amount. The amount of buffers can be retrieved through `api.getNumInputs()` and `api.getNumOutputs()`.

While the plugin is in an activated state, APE may call the following function asynchronously:

```
Status _cdecl OnEvent(CProject * p, CEvent * e);
```

This function is not required to be supported as it may be overkill for simple functions. The compiler is allowed to return `Status::STATUS_NOT_IMPLEMENTED` immediately. If it is to be supported, implementation details of `CEvent` can be found in *src/APE/APE/APESupport.h*. Use of events is described in *examples/events.h*. Supposed return value is either `Status::STATUS_OK` or `Status::STATUS_HANDLED`.

When the plugin is put in a deactivated state, the following function is called (synchronously, ie. never while an event is handled or while the plugin is processing):

```
Status _cdecl DisableProject(CProject * p);
```

This function ensures correct termination of the script and runtime, and puts the compile code in a deactivated state, from which it may be activated again through `ActivateProject()`.

When the project is no longer used, the following function is called:

```
Status _cdecl ReleaseProject(CProject * p);
```

This function shall ensure correct cleanup of the compiled code and any associated resources the compiler has allocated. The module the compiler resides in may not be used again before it is unloaded, so it's important to release everything that won't be destroyed automatically in this function.

The following function are not yet required to be implemented, however they must exist:

```
Status _cdecl AddSymbol(CProject * p, const char * name, const void * mem);
Status _cdecl GetState (CProject * p);
void * _cdecl GetSymbol(CProject * p, char * s);
Status _cdecl SetErrorFunc(CProject * p, void * op, errorFunc_t e);
```

It is advised to return `Status::STATUS_NOT_IMPLEMENTED` from these functions - or NULL in case of pointers. To properly learn the interaction and process between these functions, study the provided compiler(s) and their associated scripts.

Shipped compilers

APE currently ships with *Tcc4Ape*, an open-source compiler wrapper for APE based upon the open-source C compiler *TCC*[iii]. Also APE includes a wrapper called *syswrap*, that allows to interface to system compilers.

Tcc4Ape

Tcc4Ape is the provided, standard compiler for the language C. Tcc4Ape, together with */includes/CInterface.h/* provides the foundation for language, which consists of C mixed with some preprocessor definitions for some syntactic sugar.

Script structure

The program structure is built like an object (with good reasoning), where you have a constructor (`onLoad`), a destructor (`onUnload`), an optional event-handler (`onEvent`) and a processor function (`processReplacing`). The typical script looks like this:

```
#include <CInterface.h>

struct PluginData
{

};

GlobalData("template.c");

enum Status onLoad()
{
	return status.ready;
}

enum Status onUnload()
{
	return status.ok;
}

enum Status processReplacing(VstFloat ** in, VstFloat ** out, VstInt32 sampleFrames)
{
	for(unsigned i = 0; i < sampleFrames; ++i)
	{
		out[left][i] = in[left][i];
		out[right][i] = in[right][i];
	}
	return status.ok;
}

enum Status onEvent(struct eventInfo * e)
{
	return status.ok;
}
```

Nothing here should look too unfamiliar to those who have worked with audio-plugins before. The `PluginData` struct is where you declare all of your instance variables, as this will be passed to all of these functions. The specific instance is called `_this`. Dereferenced alias is `self`:

```
#define self (*_this)
```

Along with `_this`, another variable is passed: `iface`. `iface` holds pointers to all the API functions delivered by APE. Dereferenced alias is `api`:

```
#define api (*iface)
```

TCC API Function Reference

- `float api.getSampleRate();`

Returns the host's current sample rate as a float.

- `int api.printLine(unsigned color, const char * fmt, ...);`

Prints a line to the associated console with the color given. Works like `printf`. Returns number of characters written. Color can be any of the following:

```
color.black
color.grey
color.blue
color.green
color.red
```

- `int api.msgBox(const char * txt, const char * title, int style, int nBlocking);`

Presents a classic message-box with the given text, title and style. Style is a bitmask which can be a combination of the following two groups (using the `mbs` (message-box-style) constant struct):

```
mbs.icon.stop                   // msgbox carries a stop icon
mbs.icon.question               // msgbox carries a question mark
mbs.icon.info                   // msgbox carries a info icon
mbs.icon.warning                // msgbox carries an exclamation point

mbs.style.ok                    // msgbox has an OK button
mbs.style.yesnocancel           // msgbox has a yes, no and cancel button
mbs.style.contrycancel          // msgbox has continue, try again and cancel button
```

`nBlocking` shall either be 0 or 1. If it is 1 (blocking), the message-box blocks and waits for the user to choose a button. If it is 0, the function returns immediately, returning an non-usable opaque value. The message-box is spawned in another thread. There can at most be 16 simultaneous message-boxes.

If the message-box is blocking, it will return either of these values, that usually represents the button pressed:

```
mbs.button.yes
mbs.button.no
mbs.button.retry
mbs.button.tryagain
mbs.button.con
```

```
mbs.button.cancel
```

Example usage:

```
auto ans = api.msgBox("Hi there!", "Hello!", mbs.icon.info | mbs.style.yesnocancel,
true);

if(ans == mbs.button.yes)
	doSomething();
else
	doSomethingElse();
```

- `Status api.setStatus(Status new);`

Requests APE to change status to the new status. APE does not guarantee to do that, however it will return its internal state every time.

- `int api.createKnob(const char * name, float * val, int type);`

Creates a knob-control with the given name. All controls have a value between `0.0f` and `1.0f`, inclusive. The knob will update `val` whenever it changes its value (if it isn't `null`). The type can be any of the following:

```
knobType.percent     // knob displays a value from 0 .. 100%
knobType.hertz       // knob displays a value from 0 .. 8000 Hz
knobType.decibel     // knob displays a value from -62.5 dB to 0 dB
knobType.fpoint      // knob displays a value from 0.0 to 1.0
knobType.ms          // knob displays a value from 0 to 1000 ms
```

Please note the display value has no influence on its internal value (`0.0` .. `1.0`). See *events* section for options on changing value and display text to arbitrary values. The return value is a tag that uniquely identifies that control.

- `int api.createRangeKnob(const char * name, const char * unit, float * val, scaleCB formatFunc, float _min, float _max);`

Creates a knob-control with the given name. The display value is formatted using the callback function scaleCB, which allows to create a scale over the range:

```
typedef float (APE_API_VARI * scaleCB)(float value, float _min, float _max);
```

`value` is the knobs internal value from `0 .. 1`. `_min` and `_max` are the original parameters passed to `createRangeKnob()`. The knob-control will use the return of this function as the display value and suffix it with the `unit` parameter. However, as with the other knob-functions, `val` will contain the knobs internal value, from `0 .. 1`.

The idea is to provide a custom range and apply a scale on it. For instance, when displaying a frequency scale in hertz, it would be logical to create a exponential scale (since human hearing is roughly logarithmic). You can provide your own callback function, but APE includes a couple:

```
scale.log()
scale.exp()
scale.linear()
scale.polyLog()
scale.polyExp()
```

These can be passed directly to the function, like so:

```
auto tag = api.createRangeKnob("Frequency", "Hz", &self.value, scale.exp, 10, 1000);
```

You can also use these casually:

```
float scaledValue = scale.log(1, 10, 1000); // scaledValue is 1000
```

Please note that for `scale.log` and `scale.exp`, `_min` has to be above zero (anything else wouldn't make mathematical sense). If you need this functionality, consider `scale.polyLog` or `scale.polyExp`, which provides fixed curves based on polynomials.

- `timer api.timerGet();`

Returns an opaque handle that represents a starting point in time.

- `double api.timerDiff(timer start);`

Returns the difference in time from a previous call to `api.timerGet()` in milliseconds.

- `void * api.alloc(size_t size);`

Allocates a block of zero-initialized memory of `size` size in bytes. Memory will be deallocated automatically when the your code is destructed. Do not release memory allocated through `api.alloc()` with anything else than `api.free()`.

- `void api.free(void * ptr);`

Frees a block of memory from a previous call to `api.alloc()`. Do not free memory from other sources with this function.

- `int api.createKnobEx(const char * name, float * val, const char * values, const char * unit);`

Creates a knob-control that will display its value using the |-seperated list given in `values` and suffix it with the given `unit`. Example usage:

```
const char * values = "left|center|right";
auto tag = api.createKnobEx("pan", &self.pan, values, "pan");
```

Return value is a tag.

- `void api.setInitialDelay(int samples);`

Informs the host that the plugin delays the audio stream by `samples` amount. Useful when creating anything with lookahead. Note that the host may ignore this call, and is not obliged to perform delay compensation.

- `int api.createLabel(const char * name, const char * fmt, ...);`

Creates a label-control with the given `name`. Its display value depends on the format string given. The `fmt` string describes a `printf`-like syntax, but the additional parameters are pointers to the elements. Example:

```
char * s;
int var;
float result;

auto tag = api.createLabel("Test result", "%d, %f, %s", &var, &result, &s);

while (true) {
        var = rand() % 20;
        if(var > 10) {
                        result = var * M_PI * 2;
                        s = "succes";
        }
        else
        {
                        result = 0;
                        s = "failure";
        }
}
```

The label will then automatically display the values concurrently. Here are the supported format specifiers and what they correspond to:

%x	%u	%i	%d	%f	%lf	%c	%s	%ld	%lu	%li
void *	unsigned int	int	int	float	double	char	char *	long double	unsigned long long	signed long long

`api.createLabel()` does not support precision-specifiers or anything like that. Double `%%` yields a single `%`. Remember that doubles and floats are seperate specifiers, because normally in C floats are automatically promoted to doubles. This conversion does obviously not happen when you pass a pointer to a float, therefore the difference.

Note that, especially for strings, you must change the pointer atomically (eg. `strcpy`'ing directly on the pointer might lead to a crash). Also make sure the pointers are valid throughout your code's lifetime.

- `int api.getNumInputs();`

Returns the current number of input channels.

- `int api.getNumOutputs();`

Returns the current number of output channels.

- `int api.createMeter(const char * name, float * val);`

Creates a meter-control with the name `name`. This control shows a meter representing the `val`. Returns a tag.

- `int api.createToggle(const char * name, float * val);`

Creates a toggle control with the name `name`. Updates the value `val` points to with its internal value, if `val` isn't `null`. Return a tag.

- `double api.getBPM();`

Returns the host's tempo in beats per minute as a double.

- `float api.getCtrlValue(int tag);`

Returns the value of the control, which is uniquely identified by the `tag`.

- `void api.setCtrlValue(int tag, float val);`

Updates the control, identified by `tag` with the new value `val`.

- `int api.createPlot(const char * name, float * valList, unsigned size);`

Creates a plot of the values in the `valList`, that is, it interconnects all of the values into a graph. The values are interpreted as y values in a graph, with x being incremented by one each value. `size` is the amount of floats in the list.

User-defined functions and instance pointers

To use `api` and `self` in your own function, they will have to accept these parameters:

```
int myFunc(struct PluginData * this, struct CSharedInterface * iface) {
        self.timer = api.timerGet(); // works like normal
        this->timer = iface->timerGet(); // can also do this
}
```

You call the function like so:

```
...
myFunc(this, iface);
...
```

Macros and functions

`log2(x)`

Returns the base-2 logarithm.

`isbadf(x)`

Tests a float for +/- infinity, quiet and signaling `NaN`.

`round(x)`

Rounds the value to the nearest integer.

`sgn(x)`

`Signum` function, return -1, 0 or 1 depending on sign of function.

`ArraySize(x)`

Returns the size of the array (it has to be known at compile-time, no pointers!)

`if_sin(angle, result)`

Calculates the sine to the `angle` and sets `result` to that value. Result should be an `lvalue` of floating-point type. This is an inline parabolic approximation of `sin(x)`, that is around 15 times faster than standard library `sin(x)`. The domain of this function is $[-\pi, \pi]$. Naturally, it is less precise. Beware that it is two lines of code, so this example is illegal:

```
if(something)
	if_sin(angle, result);
```

Do this instead:

```
if(something) {
	if_sin(angle, result);
}
```

`float f_fmod(float x)`

Fast version of the library `fmod()` function.

Types

```
typedef unsigned char bool;
```

Assumes a boolean value (`true` and `false` is also declared constants `1` and `0`).

```
typedef long long timer;
```

Holds opaque timer values.

```
struct PluginData;
```

A user-defined struct which is passed to the four normal functions.

```
typedef int VstInt32;
```

An integer type compatible with VST-platforms.

```
typedef ... VstFloat;
```

A floating point type with a precision matching the host (eg. double for 64-bit signal paths).

syswrap

syswrap acts as another layer of indirection. It will run a couple of shell scripts upon the various events described in the compiler api. It expects these scripts to create a compatible dynamic library in the same folder, that syswraps loads and installs in the signal chain. This means that you can use any compiler installed on your system for any language, and APE can run it directly.

Scripts

There are three scripts located in the `/compilers/syswrap/` directory. You will have to modify these scripts to your own system. The type of script will change depending on operating system, but here's an example for Windows (which runs batch files, *.bat) and `cl.exe`, the Visual Studio C/C++ compiler.

environment script

The environment script is run when the syswrap instance is run the first time. It's intended use is to prepare the system/compiler/environment to compile commands, however it doesn't necessarily have to contain anything.

build script

The build script is run when the user wants to compile the code. The purpose of this script is therefore to turn the input files into a dynamic library. syswrap passes the expected output file name as the first commandline parameter, and a single string of input files, compatible with the current system's command line syntax, as the second parameter. The build script could look like this:

```
@echo off
call "C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\vcvarsall.bat"
cl.exe /I "C:\Audio\VstPlugins\APE\includes" /arch:SSE2 /Oi /O2 /Ot /GL /MT
/fp:precise /Gd %2 /link /DEF:"C:\Audio\VstPlugins\APE\compilers\syswrap\syswrap.def"
/DLL /OUT:%1
```

In this example, the command `@echo off` disables line debugging in the console to avoid verbose spam. It then runs the file `vcvarsall.bat` which sets up the environment for the visual studio compiler. Then it runs the visual studio compiler - here's a rundown of the arguments given to the compiler:

```
cl.exe ::
/I "C:\Audio\VstPlugins\APE\includes" -> specifies a include directory to search in
/arch:SSE2 -> specifies it should compile the program with SSE extensions
/Oi -> expands intrinsic functions that can speed up the program
/O2 -> enables optimization level 2
/Ot -> favors fast code, but may create larger code
/GL -> optimizes whole program
/MT -> statically links the program with the C runtime
/fp:precise -> floating point operations must be precice
/Gd -> uses _cdecl calling convention as default
%2 -> passes all input files in second commandline argument
/link -> following switches are for the linker
/DEF:"C:\Audio\VstPlugins\APE\compilers\syswrap\syswrap.def" -> specify a .def file
to generate correct symbols, see next section
/DLL -> create a dynamic library
/OUT:%1 -> specify output file name using first commandline argument
```

cleanup script

The cleanup script is run when syswrap is unloaded, and allows a clean up process (for instance, the system compiler may have created several intermediate files in the directory) where you can delete the generated files. syswrap passes the output file name (without extension!) as the first commandline argument. It could look like this:

```
@echo off
echo cleaning up files...
set "path1=%1%"
set "path2=%path1%.dll"
del "%path2%"
set "path2=%path1%.exp"
del "%path2%"
set "path2=%path1%.lib"
del "%path2%"
```

This will clean up all files the Visual Studio compiler created with the previous arguments.

syswrap compiler API

After the scripts have run successfully, syswrap tries to load the dynamic library it specified to the build script. If successful, it expects the following function to be exported:

```
struct _program_info * getProgramInfo()
```

This function returns a pointer to a filled `_program_info struct`, that determines how syswrap should interface with the program, and more importantly, who allocates the `PluginData` struct.

```
struct _program_info {
	size_t allocSize;
	size_t version;
	const char * name;
	int selfAlloc;
	void * (_cdecl *palloc)(struct CSharedInterface * iface);
	void * (_cdecl *pfree)(void * pluginData);
};
```

As with TCC4Ape, the four normal functions (`onEvent`, `processReplacing`, `onLoad`, `onUnload`) first two arguments are a pointer to plugin-defined instance data (`PluginData`), and a pointer to the interface API. The `PluginData` struct is defined by the plugin itself and is the object, that represents this specific instance of the plugin. The plugin can specify its own allocators for this struct (it's completely opaque to syswrap/APE), or it can specify how large it is - then syswrap will allocate and free it, and zero-initialize it.

`allocSize` is the size of the `PluginData` struct. This field is ignored if `selfAlloc` is set. `version` is the version of the plugin. `name` is the name of the script/plugin. If `selfAlloc` is nonzero, syswrap will not allocate memory for the `PluginData` struct, but instead call `palloc()` and `pfree()` on creation/destruction to allow the library to create its own object (useful if `PluginData` is non-POD).

This process and information is only relevant if you do not use the included header "CInterface.h" - if you use "CInterface.h" all of this is done transparently, provided you follow the guidelines found in the Tcc4Ape API reference. It is done this way to allow complete compability with Tcc4Ape scripts (ie. you can run Tcc4APE scripts directly using syswrap), but also to provide a C interface that does not rely on a C-header (so you can use other languages).

The following four functions shouldn't be new (see Tcc4Ape API reference).

```
enum Status onLoad(void * pluginData, struct CSharedInterface * iface)
-> onLoad
```

```
enum Status processReplacing(void * pluginData, struct CSharedInterface * iface,
float ** in, float ** out, int sampleFrames)
-> processReplacing
```

```
enum Status onEvent(void * pluginData, struct CSharedInterface * iface, struct
eventInfo * e)
-> onEvent
```

```
enum Status onUnload(void * pluginData, struct CSharedInterface * iface)
-> onUnload
```

Because of how syswrap expects these to be exported, it might be useful to specify to the compiler that these functions should be exported as-is. This can be done with Visual Studio compiler by using a module-definition-file (*.def), and a compatible one would look like this:

```
EXPORTS
        processReplacing=processReplacing
        onLoad=onLoad
        onUnload=onUnload
        onEvent=onEvent
        getProgramInfo=getProgramInfo
```

To specify that APE should use the syswrap compiler for certain languages, just edit the config file accordingly (see the section).

Changelog

- Alpha 0.3.0: 08-04-2014
    - Source code rewritten to support JUCE also, which is the primary target platform now.
        - this affects several things; notably the editor is switched from Scintilla to JUCE's inbuilt code editor now
            - this has the welcome sideeffect of hotkeys working again
        - syntax highlight only for C++ and friends for the moment.
    - x32 and x64 builds on both Windows and OSx as AudioUnit and VST 2.4
    - Countless bug fixes / code rewrite
    - project recall now implemented
    - autosave now implemented
    - support for high dpi display
    - an actual threading- and multi-instance model is now implemented; it should be completely safe to run multiple plugins in the same or other processes
    - fix of fpu exceptions
    - improved header support for other compilers than tcc
- Alpha 0.2: 10-02-2014
    - fixed uninitialized variable 'Engine::clocksPerSample'
    - scilexer now properly adds filenames to project struct even in case of singleString-compilation
    - scilexer now properly sets amount of files in the project
    - the console should now properly print strings with linebreaks in them, this affects the core, api and scripts.
    - fixed a bug where newlines will crash the console code.
    - output logging of console now properly contains newlines.
    - due to larger amounts of info being printed to the console, it is now scrollable and has a longer history
    - added new compiler: syswrap. syswrap allows to interface to installed system compilers.
    - fixed a bug where closing the editor would not reset the editor button in APE
    - pressing the editor button now properly restores the window if user had minimized it before
    - fixed a memory leak in the TCC compiler (early return caused no deallocation of plugin data)

- fixed a wrong return value in CInterface.h
- added a new knob function: api.createRangeKnob(). This knob formats it's display value based off a minimum, maximum and a callback function.
- fixed a bug where knobs initially would have the wrong format
- to enhance c++ compability, 'this' is now an illegal identifier
- CInterface.h:
  - added new valuestruct: scale
  - added f_mod() and f_sin()
  - added pi values
  - compatized header with various compilers
- updated the example scripts to reflect these changes.

Alpha 0.1

- Initial release.

Licenses

See /licenses/.

Credits and thanks

Thanks to the helpful community at *kvraudio*[iv] - extremely helpful resource and excellent site.

Thanks to *stackoverflow*[v] - always helping with design / coding issues

Thanks to *musicdsp*[vi] - besides being a hugely helpful site with many examples, it inspired me, this project and delivered source code to several of the included plugin examples.

Thanks to innovators and coders of Scintilla, TCC, VSTGUI[vii], libconfig, JUCE[viii] and Steinberg[ix] for delivering incredibly nice products that without doubt makes projects like mine realizable.

Contact

Did you create a cool effect you want to share and possibly include in further releases of APE? Need support? Got inquiries about the product? Have some feedback/suggestions? Any violations I need to know about? I can be contacted at the following email:

dyanuzz@hotmail.com

Donations

I study full time and use most of my free time developing free, open-source tools like these. If you want to show your appreciation, you are very welcome to donate to the following PayPal:

dyanuzz@hotmail.com

If nothing else is noted, I will split the donations 50/50 to me and the other half to the rest of the free, open-source projects that made this project possible.

Thank you.

Janus Lynggaard Thorborg

studying bachelor's degree in sonic communication and sound design

Sonic College, Haderslev in Denmark

---

[i] scintilla: http://www.scintilla.org/
[ii] libconfig: http://www.hyperrealm.com/libconfig/
[iii] tiny c compiler: http://bellard.org/tcc/
[iv] kvraudio: http://www.kvraudio.com/
[v] stackoverflow: http://stackoverflow.com/
[vi] musicdsp: http://www.musicdsp.org/
[vii] vstgui: http://sourceforge.net/projects/vstgui/
[viii] juce: http://www.juce.com
[ix] steinberg: http://www.steinberg.net